Sunday
Date 07/31/2022

Lecture 30 :- Declaration & Access modifiers.

(1) Java Source file Structures.
(2) Class Level modifiers
(3) Member Level modifiers
(4) Interfaces.

(1) Java Source file Structure :-

A Java program can contain any no. of classes but atmost One class can be declared as public. If there is a public class then name of the program and name of the public class must be matched otherwise we will get CTE.

eg :-1 :-

```
class A
{
}
class B
{
}
class C
{
}
```

← Java source file.

Case 1: If there is no public class then we can use any name and there no restrictions. eg :- A.Java, B.Java,

C.java, durg.java.

Case 2: If class B is public then name of the program should be B.java. otherwise we will get CTE: class B is public, should be declared in a file named B.java.

Case 3:

```
class A
{
}
public class B
{
}
public class C
{
}
```

Java source file

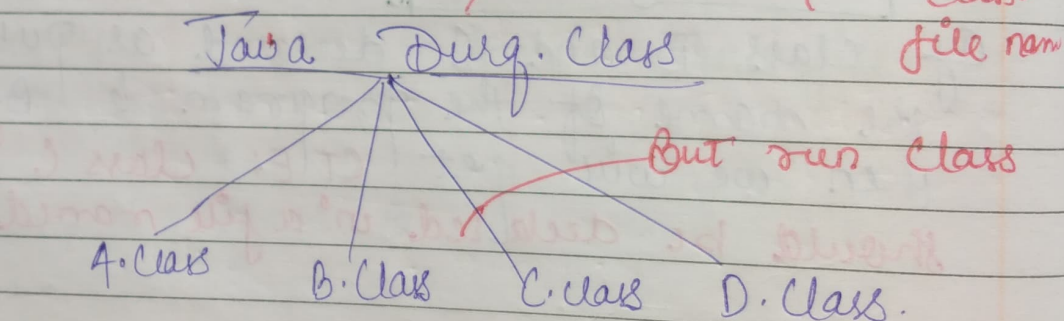
If class B and C declared as public, and the name of the program is B.java. then we will get CTE: class C is public, should be declared in a file named C.java.

eg-2:

```
class A
{
    p s v main(String[] args)
    {
        sopln("A class main");
    }
}
class B
{
```

Durg.Java

```
{
class A
{
    P S V main(String[] args)
    {
        sopln("A class main");
    }
}
class C
{
    P S V main(String[] args)
    {
        sopln("C class main");
    }
}
class D
{
}
```



Java A ↵

O/p: A class main

Java B ↵

O/p: B. class main

Java C

C. class main

Java D.

RE: NoSuchMethodError: main

CLASSTIME Pg. No.
Date / /

Java Durga ↵

RE: NoClassDefFoundError: Durga.

Conclusions:

(1) Whenever we are compiling a Java program for every class present in that .java program a separate .class file will be generated.

(2) We can compile a Java program (Java source file) but can't run only a Java .class file.

(3) Whenever we are executing a Java the corresponding class main method will be executed.

(4) If the class doesn't contain main method then, we will get RE: NoSuchMethodError: main.

(5) If the corresponding .class file not available then we will get RE: NoClassDefFoundError: Correspondingclassname.

(6) It is not recommended to declare multiple classes in a single source file.
It is highly recommended to one declared only one class per source file and name of the program we have to keep same as class name. The main advantage of this approach is readability and

maintainability of code will be improved.

* import statement :-

```
class Test
{
    p s v main(String[] args)
    {
        ArrayList l = new ArrayList();
    }
```

CE: Cannot find symbol
Symbol: class ArrayList
location: class Test.

We can solve this problem by using fully qualified name.

```
p s v main(String[] args)
{
    java.util.ArrayList l = new java.util.ArrayList();
}
```

↳ fully qualified name.

The problem with uses of fully qualified name everytime is —
It increases length of the code & reduces readability.
We can solved this problem by using import statement.
Whenever we are writing import statement it is not required to use

fully Qualified name we can use short name directly.

eg:- import java.util.Arraylist;
class Test
{
    p s v main(String[] args)
    {
short name → ArrayList l = new Arraylist();
    }
}

Hence, import statement act as typing shortcut.

Lecture 31:- import & static import.

Case 1:- Types of import statements.

There are two types of import statements-

(1) Explicit class import
(2) Implicit class import.

(1) **Explicit class import:-**

eg:- import java.util.Arraylist;

It is highly recomended to use explicit class import, because it improves readability of the code. best suitable for competitive coding where readability is important.

(2) **Implicit class import:-**

eg: Java.util.*;

not recomended to use, because it reduces readability of the code. Best suitable for interpret where typing is important.

Case 2:

Q Which of the following import statements are meaningful

✓ import java.util.AL; (util: package name, AL: class name)

✗ import java.util.AL.*;

✓ import java.util.*; (java.util: package, *: for all class)

✗ import java.util; (util: package)

Note: After package name (like util) .* is allowed. or class name.

After class name (like AL) ; is required.

Case 3: Consider the following code-

class myObject extends java.rmi.UnicastRemoteObject
{
}

(myObject: class name, extends: keyword, java.rmi.UnicastRemoteObject: fully qualified name.)

The code compiles fine even though we are not writing import statements, because we used fully qualified name.

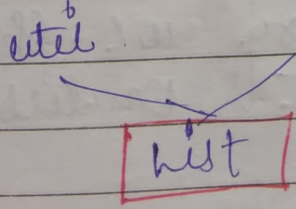
Note: Whenever we are using fully qualified name it is not required to write import statement, similarly whenever we are writing import statement it is not req to use fully qualified name.

Case 4:

```
import java.util.*;
import java.sql.*;

class Test
{
    p s v main(String[] args)
    {
        Date d = new Date();
    }
}
```

CE: reference to Date is ambiguous.

Note: Even in the case of List also we may get same ambiguity problem, because it is available in both util & Awt Package.



CLASSTIME Pg. No.
Date / /

Case 5 : While resolving class names compiler will always gives the precidence in the following order.

1. Explicit class import
2. classes present in CWD (current working directory) (default package)
3. Implicit class import

eg : import java.util.Date;
import java.sql.*;
class Test
{
psvm(String[] args)
{
Date d = new Date();
sopln(d.getClass().getName());
}
}

In the above eg. util package date got considered.

Case 6 : Whenever we are importing a java package all classes & interfaces present in that package by default available. but not sub-package classes. If we want to use subpackage class compulsory we should write import statement until subpackage level.

eg: Java
- util
  - regex
    - Pattern → class
    - Matcher → class

To use Pattern class in our program which import statement is required?

1. import java.*;
2. import java.util.*;
3. import java.util.regex.*;
4. No import required.

Case 7: All classes and interfaces present in the following packages are by default available to every java program. Hence, we are not required to write import statement.

1. java.lang. — package
2. default package (CWD)

eg:
```
class Test
{
    P S V main(String[] args)
    {
        String s = new String("durga");
        Student s1 = new Student("durga", 101);
        System.out.println(s1.name + "..." + s1.rollno);
    }
}
```
java.lang. → String
CWD → Student, s1, System

CLASSTIME Pg. No.
Date / /

Case 8:- import statements is totally compile time related concept. If more no. of imports then more will be the CT. But there is no effect on execution time. (Runtime).

Fully Qualified name.
java.util.AL

CT ↓ min

import stam;
short name
AL

CT ↑ max

Case 9:- Diffence b/w C lang. #include and Java lang. import statement.

In the case of C lang. #include all input/output header file will be loaded at the beginning only (at translation time) Hence it is static include.

But in the case of Java import statement no .class file will be loaded at the beginning. whenever we are using a particular class then only corresponding .class file will be loaded. this is like dynamic include. or load on Demand (or) load on fly.

Note

2)
3)
4)
5)
6)
7)
8)
9)

(9)

Note: 1.5V new features.

1) for-each loops
2) var-args methods
3) Autoboxing & Autounboxing
4) Generics
5) Co-variant return types
6) Queue
7) Annotations
8) enum
9) Static import.

} ✓ bit concept

} ✗ not recomended for us

(9) Static import: Introduced in 1.5 VN.

Acc to SUN Microsystem Use of static import reduces length of the code and improves readability. but acc to worldwide programming experts (like us) uses of static import creates confusion and reduces readability. Hence if there is no specific requirement then it is not recommended to use static import.

Usually we can access static members by using class name. But whenever we are writing static import we can access static members directly without class name.

Eg:

Without static import :-

```
class Test
{
    p s v main(Str q--)
    {
        sopln(Math.sqrt(4));
        sopln(Math.max(10,20));
        sopln(Math.random());
    }
}
```

class — Math
method name — sqrt

With static import.

```
import static java.lang.Math.sqrt;
import static java.lang.Math.*;

class Test
{
    p s v main(Str--)
    {
        sopln(sqrt(4));
        sopln(max(10,20));
        sopln(random());
    }
}
```

class name — Math
method — sqrt

Lecture 32 :-

Explain about System.out.println.

eg :-

```
class Test
{
  static String s = "java";
  :
}
```

Test.s.length()

- "Test" is a class name.
- "s" is a static variable present in Test class of the type of java.lang.String
- length() is a method present in String class.

```
class System
{
  static PrintStream out;
  :
}
```

System.out.println()

- "System" is a class present in java.lang package.
- "out" is a static variable present in System class of the type PrintStream.
- "println()" is a method present in PrintStream class.

Out is a static variable present in System class. Hence we can access by using class name System, Out when ever we are writing static import it is not required to use class name and we can access out directly.

eg:-

```
import static java.lang.System.Out;
class Test
{
    p s v main(String[] args)
    {
        Out.println("Hello");
        Out.println("Hi");
    }
}
```

O/p:- Hello
Hi.

Que- Why dont we apply static import for some other case also.

Ans- eg:-

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;

public class Test
{
    p s v main(String[] args)
    {
        sop(MAX_VALUE);
    }
}
```

(E). reference to MAX_VALUE is ambiguous.

While resolving static members compiler will always consider the precedence in the following order.

(1) Current class static members (1 & 2)
2) Explicit static import
implicit static import.

Eg: import static java.lang.Integer.MAX_VALUE; → (2).
import static java.lang.Byte.*;

```
public class Test
{
    static int MAX_VALUE = 999;  — (1)
    public static void main(String[] args)
    {
        sopln(MAX_VALUE);
    }
}
```

O/p: 999

⇒ If we comment line 1 then explicit static import will be considered and hence Integer class max_value will be considered. In this case the Output is 2147483647.

⇒ If we comment both lines (1) & (2) then implicit static import will be considered and hence Byte class MAX_VALUE will be considered. In this case Output is 127.

CLASSTIME Pg. No.
Date / /

① Normal import

①a Explicit import:

Syntax:

import package name. Class name;

eg:

P.N

import java.util. Arraylist;

①b Implicit import:

Syntax:

import packagename. *;

eg:

P.N.

import java.util. *;

② Static import:

②a Explicit Static import:

Syntax:

import static packagename. Classname. Static member;

eg:

P.N Class S.M (var)

import static java.lang. Math. sqrt;

import static java.lang.System.out;

(b) Implicit static import:-

Syntax:

import static packagename.classname.*;

eg:

P.N C.N

import static java.lang.Math.*;
import static java.lang.System.*;

Q. Which statement is true/valid?

import java.lang.Math.*; ✗
import static java.lang.Math.*; ✓
import java.lang.Math.sqrt; ✗
import static java.lang.Math.sqrt(); ✗
import java.lang.Math.sqrt.*; ✗
import static java.lang.Math.sqrt; ✓
import java.lang; ✗
import static java.lang; ✗
import java.lang.*; ✓
import static java.lang.*; ✗

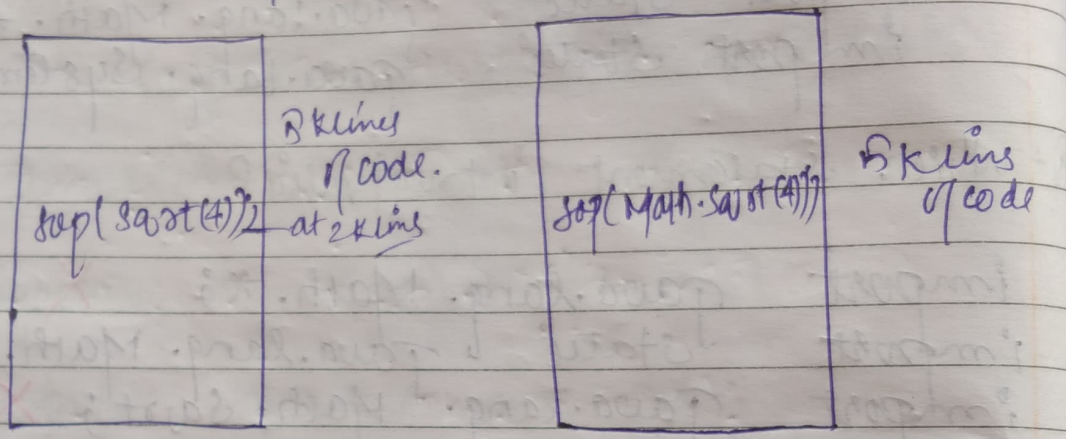
Two packages contains a class or interface with the same name is very rare. Hence, ambiguity problem is also very rare in normal import.

But two classes or interfaces contain a variable or method with the same name is very common, hence ambiguity problem is

CLASSTIME Pg. No.
Date / /

also very common problem. in static import.

Uses of static import reduces readability and creates confusion. Hence, if there is no specific requirement then it is not recommended to use static import.



Difference b/w Normal import and static import:-

We can use normal import to import classes and interfaces of a particular package.

Whenever we are using normal import it is not required to use fully qualified name and we can use short names directly.

We can use static import to import static members of a particular class or interface.

CLASSTIME Pg. No. Date / /

Whenever we are writing static import if is not required to use classname to access static members and we can access directly.

Lecture 33 :- Packages / Package Statement :-

Package :- It is a encapsulation mechanism to group related classes and interfaces into a single unit, which is nothing but package.

eg1 :- All classes and interfaces required for database operations are grouped into a single package called java.sql package.

eg2 :- All classes & interfaces which are grow useful for file IO operations are grouped into a seperate package which is nothing but java.io package.

The main advantages of package are :-

1) To resolve naming conflicts (i.e unique identification for components)
2) Modularity Improved. of applicn
3) It improves maintainability of appln.
4) It provides security for out components.

There is one Universally accepted naming convention for packages - i.e To use internet domain name in reverse.

eg :-

com.icicibank.loan.housing.Account.

- com → clients internet domain name in reverse.
- icicibank → module name
- loan → submodule name
- housing → class name
- Account → class name

eg:-

```
package com.durgasoft.scjp;
public class Test
{
    p s v main(String[] args)
    {
        sop("pkg demo");
    }
}
```

① Javac Test.java ↵

generated .class file will be placed in corresponding package structure in CWD.

CWD
└ Test class

② Java -d . Test.java ↵

- -d → destination to place generated .class files
- . → CWD

CLASSTIME Pg. No.
Date / /

Generated class file will be placed in corresponding package structure.

CWD
- com
  - durgasoft
    - scjp
      - Test.class.

⇒ If the corresponding package structure not already available then this command itself will create corresponding package structure.

⇒ As destination instead of . we can take any valid directory name.

eg: javac -d F: Test.java.

F:
- com
  - durgasoft
    - scjp
      - Test.class.

If the specified directory not already available then we will get CTE:

eg: javac -d T: Test.java.

if z: not available then we will get
CTE: directory not found: z:

At the time of execution we have to use fully qualified name.

eg java.durgasoft.sjp.Test

Op- package demo.

Conclusion 1:-
In any java source file there can be atmost one package statement.
i.e. more than 1 package statement is not allowed otherwise we will get
CTE: class, interface or enum expected.

```
package pack1;
package pack2;
Public class A
{

}
```

Conclusion 2:-

```
import java.util.*;
```

In any java program the first non-comment statement should be package statement (if it is available). Otherwise we will

CLASSTIME Pg. No.
Date / /

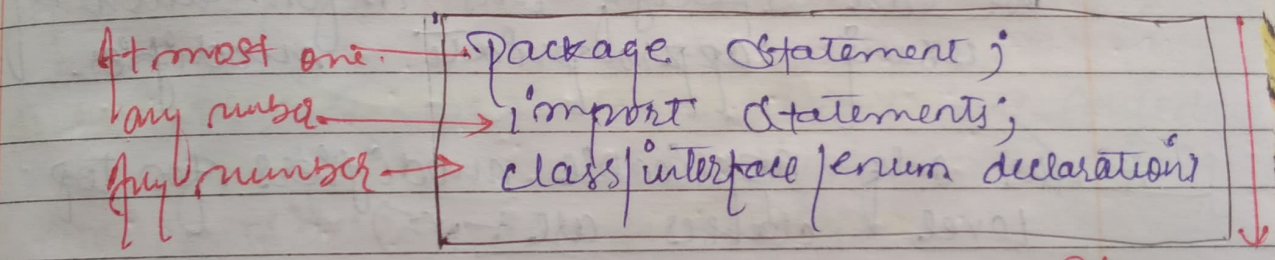
get CTE:

eg:

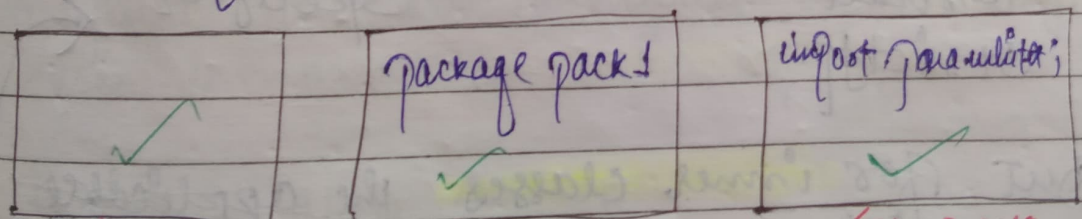
```
import java.util.*;
package pack1;
public class A
{
}
```

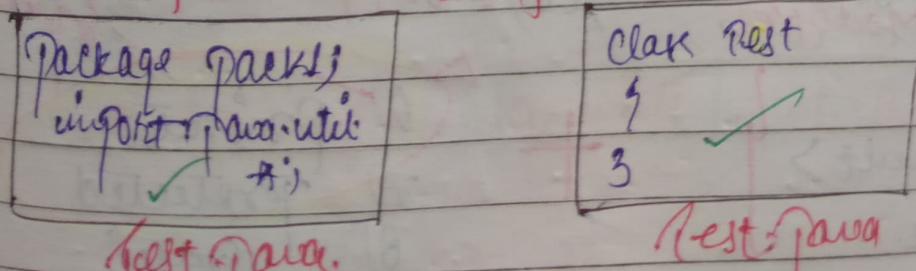
CE: class, interface, or enum expected

The following is valid java source file structure.



Note: An empty source file is a valid java program. Hence the following are valid java source files.





CLASSTIME Pg. No.
Date / /

Lecture 34 :- (2) class Level Modifiers :-

Whenever we are writing our own classes we have to provide some information about our class to the JVM. like -

1) whether this class can be accessible from anywhere or not.
2) whether child class creation is possible or not
3) whether object creation is possible or not etc.

we can specify this information by using appropriate modifier.

★ The only applicable modifiers for top level classes are :- (5)

1) public
2) <default>
3) final
4) abstract
5) strictfp.

Specifier → in c++ & old languages in Java no word exist like specifier

But for inner classes the applicable modifiers are - (8) also top level

Member Level Modifiers.

1) public
2) <default>
3) final

+

6) private
7) protected
8) static

specifier

4) abstract
5) strictfp

eg:

```
private class Test
{
    p s v main(String[] args)
    {
        Sop("hello");
    }
}
```

CE: modifier private not allowed here

* Access specifiers Vs Access modifiers:

public, private, protected, default are considered as specifiers, except these remaining are considered as modifiers.

But this rule is applicable only for old languages like C++, but not in Java.

In Java all are considered as modifiers only. there is no word like specifiers.

```
private class Test
{
}
```

CE: modifier | private | not allowed here.

(1) Public classes: If a class declared as a public then we can access that class from anywhere.

Eg:

```
package pack1;
public class A
{
    public void m1()
    {
        sopln("Hello");
    }
}
```

src.file 1.

javac -d . A.java

pack1
|— A.class.

```
package pack2;
import pack1.A;
class B
{
    p s v main(String[] args)
    {
        A a=new A();
        a.m1();
    }
}
```

src file 2

javac -d . B.java.
java pack2.B.

CLASSTIME Pg. No.
Date / /

O/p: Hello.

If class A is not public then while compiling B class we will get CTE! Pack1.A is not public in pack.1; Cannot be accessed from outside package.

(2) default classes: If a class declared as default then we can access that class only within the current package. i.e from outside package we can't access. Hence, default access is also has package level access.

(3) final modifiers: final is a modifier applicable for classes, methods and variables.

(1) final method: Whatever methods parent has by default available to the child through inheritance. If the child not satisfied with parent method implementation then child is allowed to redefine that method based on its requirement. This process is called overriding.

So, If the parent class method is declared as final then we can't override that method in the child class because its implementation is final.

eg:

```
class P
{
    public void property()
    {
        sopln("Cash+Land+Gold");
    }
    final
    public void marry()
    {
        sop("Subalanni");
    }
}

class C extends P
{
    public void marry()
    {
        sopln("xuy|pasi|zuy");
    }
}
```

overridden method.

means inherit P

overriding method

CE: marry() in C. Cannot override marry() in P; overridden method is final.

(3)(2) final class :- If a class declared as final we can't extend functionality of that class. i.e. we can't create child class for that class. i.e. inheritance is not possible for final classes.

CLASSTIME Pg. No.
Date / /

```
final P
{
}
class C extends P
{
}
```

↳ C cannot inherit from final P

* Note: Every method present inside a final class is always final by default/Implicitly but every variable present inside final class need not be final.

eg:
```
final Class P
{
    static int x = 10;
    public static void main(String[] args)
    {
        x = 777;
        System.out.println(x);
    }
}
```

O/P ⇒ 777

Note: The main advantage of final keyword is we can achieve security and we can provide unique implementation.
But the main disadvantage of final keyword is we are missing key benefits of OOPs. like - Inheritance (Because of final classes).

and - polymorphism (Bec. of final methods).
Hence if there is no specific requirement then it is not recommended to use final keywords.

Lecture-35 :- Abstract Modifier :-

(4) Abstract is a modifier applicable for classes and methods but not for variables.

(4)(a) Abstract method :- Even though we don't know about implementation still we can declare a method with abstract modifier. i.e. for abstract methods only declaration is available but not implementation. Hence, abstract method declaration should ends with ";"

eg :- public abstract void m1(); ✓
public abstract void m1(){} ✗

child class is responsible to provide implementation for parent class abstract methods.

eg :- abstract class Vehicle
{
abstract public int getNoOfWheels();
}

```
class Bus extends vehicle
{
  public int getNoOfWheels()
  {
    return 7;
  }
}
class Auto extends Vehicle
{
  public int getNoOfWheels()
  {
    return 3;
  }
}
```

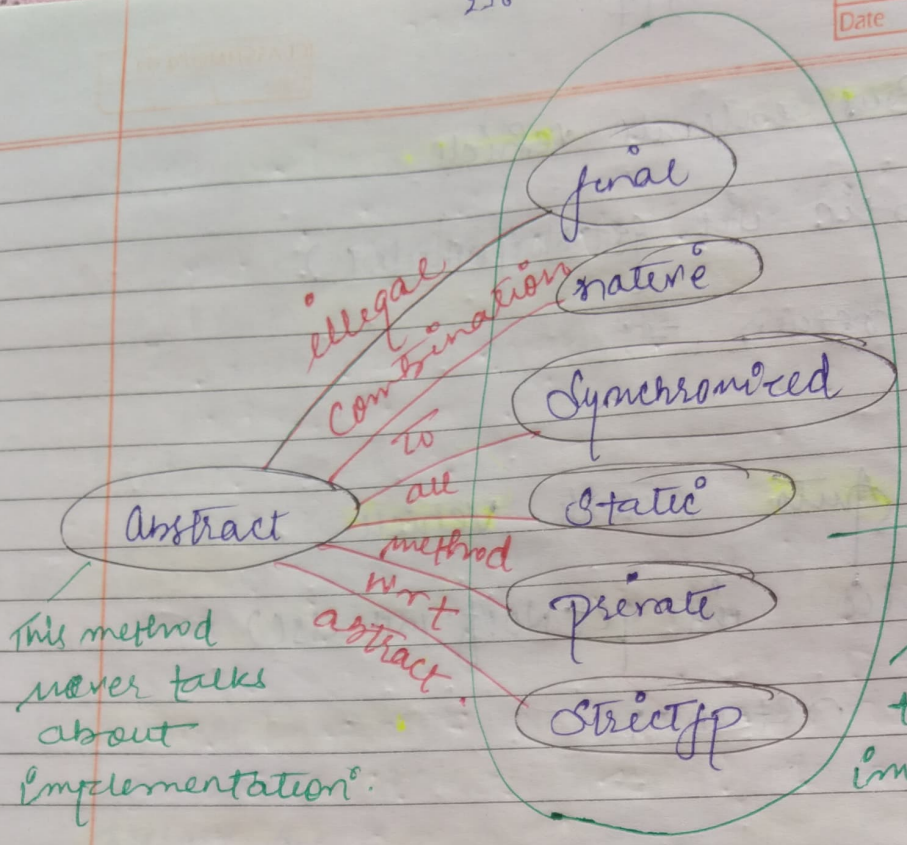
By declaring abstract method in the parent class we can provide guidelines to the child classes such that which methods compulsory child has to implement.

Abstract method never talks about implementation if any modifier talks about implementation then it forms illegal combination with abstract modifier.

The following are various illegal combinations of modifiers for methods w.r.t abstract.

eg: abstract final void m1();

CE: illegal combination of modifiers: abstract and final.



(4)(2)* Abstract class :-

For any Java class if we don't want/not allow to create an object (Because of partial implementation) such type of class we have to declare with abstract modifier. i.e. for abstract classes instantiation is not possible.

Eg:-

```
abstract class Test
{
    public s v main(String[] args)
    {
        Test t = new Test();
    }
}
```

O/P: Test is abstract; Cannot be instantiated.

abstract class Vs abstract method.

i) If a class contains atleast one abstract method then compulsory we should declare class as abstract. Otherwise we will get CTE.
Reason: If a class contains atleast one abstract method the implementation is not complete and hence it is not recommended to create object. To restrict object instantiation compulsory we should declare class as abstract.

ii) Even though class doesn't contain any abstract method still we can declare class as abstract if we don't want instantiation i.e. abstract class can contain zero no of abstract methods also.

eg: HttpServlet class is not abstract but it doesn't contain any abstract methods.

eg: Every adapter class is recommended to declare as abstract. But it doesn't contain any abstract method.

1. eg:
```
class P
{
    public void m1();
}
```
CE: missing method body, or declare abstract

CLASSTIME | Pg. No.
Date / /

2. class P
{
public abstract void m1(){}
}
CE: Abstract methods cannot have a body.

3. class P
{
public abstract void m1();
}
CE: P is not abstract and does not override abstract method m1() in P

If we are intending abstract class then for each & every abstract method of parent class we should provide implementation otherwise we have to declare that child class as abstract (also). In this case next level child class is responsible to provide implementation.

Eg: abstract class P
{
public abstract void m1();
public abstract void m2();
}
class C extends P
{
public void m1(){}
}

C2) C is not abstract and doesn't override abstract method m2 of P.

Final Vs abstract :-

1) abstract methods compulsory we should override in child classes to provide implementation. whereas we can't override final methods. hence, final abstract combination is illegal combination for methods.

2) for final classes we can't create a child class whereas for abstract classes we should create child class to provide implementation. hence final abstract combination is illegal for classes.

3) Abstract class can contain final method, whereas final class can't contain abstract method.

eg:-

```
abstract class Test
{
    public final void m1()
    {
    }
}
```

✓

```
final class Test
{
    public abstract void
        m1();
}
```

✗

Note:- It is highly recommended to use abstract modifier because it promotes several OOPs features like inheritance & polymorphisms.

CLASSTIME Pg. No.
Date / /

Lecture 36 : Member level modifier.

(5) * strictfp : [strict floating point]

1) Introduced in 1.2 version
2) We can declare/use strictfp for classes and methods but not for variables.

Usually the result of floating point arithemetic is varied from platform to platform.
If we want platform independent result for floating point arithemetic etc then we should go for strictfp modifier.

(5)(1) * strictfp method : If a method declared as strictfp all floating point calculations in that method has to follow IEEE 754 standard. So that we will get platform independent results.

abstract modifier never talks about implementation whereas strictfp methods always talks about implementation. Hence abstract, strictfp combin is illegal for methods.

(5)(2) * strictfp class : If a class declared as strictfp then every floating point cal. in present in every concrete method has to follow IEEE 754 standard. So that we will get

platform independent results.

We can declare abstract strictfp combination for classes. i.e. abstract strictfp combination is legal for classes. but illegal for methods.

```
abstract strictfp class Test
{
}
```

✓

```
abstract strictfp void m();
```

✗ C.E: illegal combination of modifiers: abstract and strictfp.

(2) Member level modifiers :- (method or variable level modifiers) :-

(1) Public members :- If a member declared as public then we can access that member from anywhere but corresponding class should be visible i.e. before checking member visibility we have to check class visibility.

Eg :-
```
package Pack1;
class A
{
  public void m1()
  {
    System.out.println("Class method");
  }
}
```
javac -d . A.java

```
package Pack2;
import pack1.A;

class B
{
    p s v main(String[] args)
    {
        A a = new A();
        a.m1();
    }
}
```

javac -d . B.java

CE: pack1.A is not public in pack1; cannot be accessed from outside of package

In the above example even though m1() is public we can't access from outside package because corresponding class A is not public. i.e. if both class & method are public then only we can access the method from outside package.

(2) default members: If a member declared as default then we can access that member only within the current package. From outside of the package we can't access. Hence default access is also called package level access.

(3)* Private members :- If a member is private then we can access that member only within that class. i.e. from outside of the class we can't access.

abstract methods should be available to the child classes to provide implementation. whereas private methods are not available to the child classes to provide implementation / hence private abstract combination is illegal for methods.

(4)* Protected members :- [The most misunderstood modifier in Java] +

If a member declared as protected then we can access that member anywhere within the current package but only in child classes of outside package.

protected = <default> + child.

We can access protected members within the current package anywhere either by using parent or child reference.

But we can access protected members in outside package only in child class and we should use child reference only.

i.e. parent reference cannot be used to access protected members from outside

CLASSTIME Pg. No.
Date / /

package.

eg:- package pack1;
public class A
{
protected void m1()
{
System.out.println("the most understood modifier");
}
}
class B extends A
{
p s v main(String[] args)
{
① A a = new A();
a.m1();

child reference

② B b = new B();
b.m1();

parent reference

③ A a1 = new B();
a.m1();
}
}

Within the package ↑

Within Current package protected member has access from anywhere either with parent class or child reference.

```
package Pack2;
import Pack1.A;
class C extends A
{
    p s v main(String[] args)
    {
        ① A a = new A();
          a.m1();  ✗

        ② C c = new C();
          c.m1();

        ③ A a1 = new C();
          a1.m1();  ✗
    }
}
```

CE: m1() has protected access in Pack1.A.

→ Outside of the package.

We can access protected members from outside package only in child classes and we should use that child class reference only. for eg:- from D class if we want to access we should use D class reference only.

```
package pack2;
import pack1.A;

class C extend A
{
}
```

CLASSTIME Pg. No.
Date / /

class D extend C
{
p s r main(String[] args)
{
① A a = new A();
a.m1();

② C c = new C();
c.m1();

③ D d = new D();
valid d.m1();

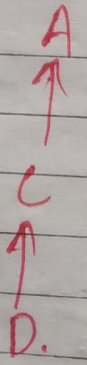
④ A a1 = new C();
a1.m1();

⑤ A a1 = new D();
a1.m1();

⑥ C c1 = new C();
c1.m1();

}
}

(C! m1() has protected access in pack1.A.



CLASSTIME Pg. No.
Date / /

Summary Table of private, default, protected and public modifier.

Visibility	private	<default>	protected	public
Within the same class	✓	✓	✓	✓
From child class of same package	✗	✓	✓	✓
From Non-child class of same package	✗	✓	✓	✓
From child class of outside package.	✗	✗	✓ We should use child reference only.	✓
From Non-child class of outside package.	✗	✗	✗	✓

Less accessible ← private < default < protected < public → more accessible

1. The most restricted access modifier is private.
2. The most accessible modifier is public.
3. Recommended modifier for datamember (variable) is private., But recommended modifier for methods is public.

CLASSTIME Pg No.
Date / /

Lecture 37 :-

* Final Variables :-

(1) Final instance variables :-

Instance variable :- If the value of a variable is varied from object to object such type of variables are called instance variables.

2. For every object a seperate copy of instance variable will be created.

3. For instance variables we are not required to perform initialization explicitly. JVM will always provide default values.

```
eg:- class Test
     {  int x;
        p s v main(String[] args)
        {
           Test t = new Test();
           sopln(t.x);   O/P :- 0
        }
     }
```

If the instance variable declared as final then compulsory we have to perform initialization explicitly wheather we are using or not and JVM won't provide default values.

CLASSTIME Pg. No.
Date / /

```
class Test
{
    final int x;
}
```

. CE! Variable x might not have been initialized.

Rule:- For final instance variable compulsory we should perform initialization, before Constructor completion.

i.e the following are possible places for initialization.

At the time of declaration:-

```
class Test
{
    final int x=10;
}
```

Inside a instance block:-

```
class Test
{
    final int x;
    {
        x=10;
    }
}
```

Inside Constructor :-

```
class Test
{
  final int x;
  Test()
  {
    x=10;
  }
}
```

These are the only possible places to perform initialization for final instance variables. If we are trying to perform initialization anywhere else then we will get CTE.

```
class Test
{
  final int x;
  public void m1()
  {
    x=10;
  }
}
```

CE: cannot assign a value to final variable x.

(ii) * final static variable :-

static variable :- If the value of a variable is not varied from object to object

Such type of variables are not recommended to declare as instance variables. we have to declare those variables at class level by using static modifier.

2. In the case of instance variables for every object a seperate copy will be created. but in the case of static variables a single copy will be created at class level and shared by every object of that class.

For static variables it is not required to perform initialization explicitly. JVM will always provide default values.

eg:
```
class Test
{
    static int x;
    p s v main(String[] args)
    {
        sop(x); // o/p: 0
    }
}
```

If the static variable declared as a final then compulsory we should perform initialization explicitly. otherwise we will get CE and JVM won't provide any default values.

ex:
```
class Test
{
    final static int x;
}
```

CE: variable x might not have been initialized.

Rule: For final static variables compulsory we should perform initialization Before class loading compilation.

i.e. the following are various places for this.

(1) At the time of declaration:

```
class Test
{
    final static int x=10;
}
```

(2) Inside static Block:

```
class Test
{
    final static int x;
    static
    {
        x=10;
    }
}
```

These are the only possible places to perform initialization for final static variables if we are trying to perform initialization anywhere else, then we will get CTE.

```
class Test
{
```

```
final static int x;
public void m1()
{
    x=10;
}
}
```

CE: Cannot assign a value to final variable x.

(iii) final Local variables: Sometimes to meet temporary requirements of the programmer we have to declares variables inside a method or block, constructor such type of variables are called local variables or temporary variables or stack variables or automatic variables.

for local variables JVM won't provide any default values compulsory we should perform initialization explicitly.

Before using that local variable i.e. if we are not using then it is not required to perform initialization for local variable.

```
class Test
{
    public static void main(String[] args)
    {
        int x;
        sopln("Hello");
    }
}
```

O/P: Hello

```
class Test
{
    p s v main(String[] args)
    {
        int x;
        sop(x);
    }
}
```

CE: variable x might not have been initialized.

Even though local variable is final before using only we have to perform initialization. i.e. if we are not using then it is not required to perform initialization even though it is final.

```
class Test
{
    p s v main(String[] args)
    {
        final int x;
        sop("Hello");
    }
}
```

O/p: Hello.

The only applicable modifier for local variable is final, by mistake if we are trying to apply any other modifier we will get CTE.

```
class Test
{
    public
    default      int x;
    private      static int y;
    protected
    p s v main(String[] args)
    {
        public final int z = 30;
    }
}
```

eg:-

```
class Test
{
    p s v main(String[] args)
    {
        public int x = 10;
        private int x = 10;
        protected int x = 10;
        static int x = 10;
        transient int x = 10;
        volatile int x = 10;

        final int x = 10;
    }
}
```

CE: illegal start of the expression.

Note: If we are not declaring any modifier then by default it is default. But this rule is applicable only for instance & static variables but not for local variables.

Formal parameters of a method :-

Formal parameter :- Formal parameters of a method simply access local variable of that method, Hence formal parameter can be declared as final ; if formal parameter declared as final then within the method we can't perform reassignment.

```
class Test
{
  p s v main(String[] args)
  {
    m1(10,20);
  }
  p s v m1(final int x, int y)
  {
    // x=100;  → CE: cannot assign a value to final variable x.
    y=200;
    sopln(x+"---"+y);
  }
}
```

Actual parameter

Formal parameters

Lecture 38 → Static modifier :-

Static is a modifier applicable for methods and variables but not for classes. We can't declare top level class with 'Static' modifier. But we can declare inner class as Static. [Such type of

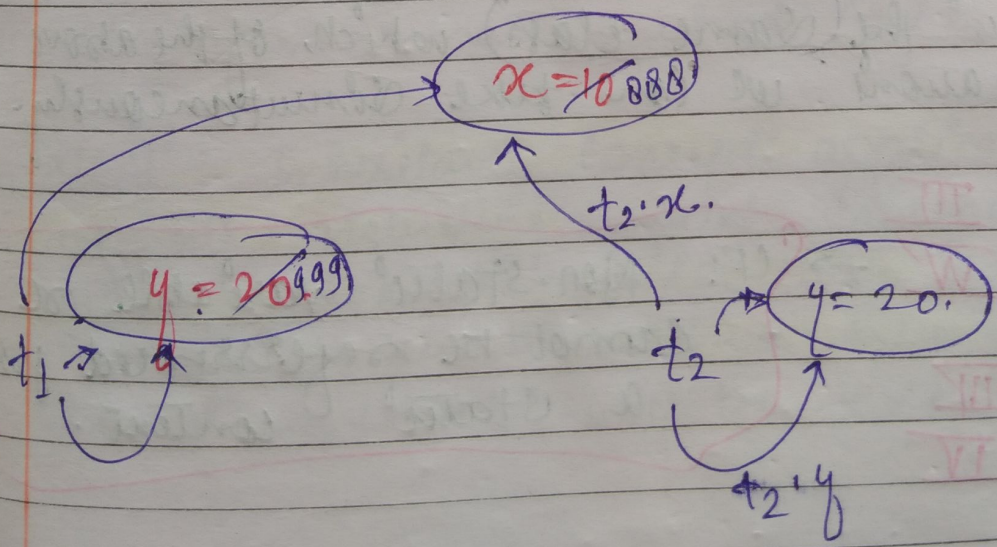
CLASSTIME Pg. No.
Date / /

innerclasses are called static nested classes].

In the case of instance variables for every object a seperate copy will be created but in the case of static variable a single copy will be created at class level and shared by every object of that class.

```
class Test
{
  static int x=10;
  int y=20;
  public static void main(String[] args)
  {
    Test t1=new Test();
    t1.x = 888;
    t1.y = 999;
    Test t2 = new Test();
    sopln(t2.x + "...." + t2.y);
  }
}
```

888 ..... 20.



We can't access instance members directly from static area, but we can access from instance area directly.

We can access static members from both instance and static area directly.

Consider the following declarations :-

I. int x = 10; → stat instance variable

II. static int x = 10; → static variable.

III. public void m1()
{
System.out.println(x);
}
} instant content / area.

IV. public static void m1()
{
System.out.println(x);
}
} static content / area.

with in the (same class) which of the above declarations we can take simultaneously.

(A) I & III ✓
(B) I & IV ✗ → CE: Non-static variable x cannot be referenced from a static content.
(C) II & III ✓
(D) II & IV ✓

(E) I & II → variable x is already defined in Test.

(E) III & IV → m1() is already defined in Test.

Case I: Overloading concept applicable for static methods including main method. but JVM can always call String[] args main method only.

eg:-

```
class Test
{
    public static void main(String[] args)
    {
        sopln("String[]");
    }
    public static void main(int[] args)
    {
        sopln("int[]");
    }
}
```

overloaded methods

O/p:- String[]

Other overloaded method we have to call just like a normal method call.

Case II:- Inheritance concept applicable for static methods including main method hence, while executing child class. if child doesn't contain main method then parent class main method will be executed.

S.E Er.CS

CLASSTIME Pg. No.
Date / /

P.java

```
class P
{
  P s v main(String[] args)
  {
    System.out.println("parent main");
  }
}
class C extends P
{
}
```

Javac P.java

P.class C.class

Java P

Op: parent main

Javac C

Op: parent main.

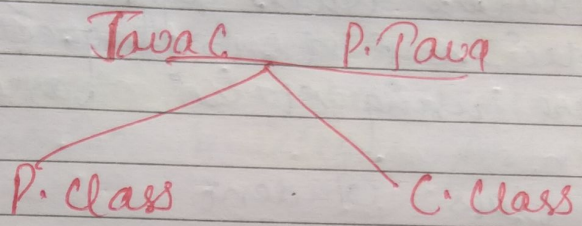
Case III :-

```
class P
{
  P s v main(String[] args)
  {
    System.out.println("parent main");
  }
}
class C extends P
```


It method hiding but not overrided

It is method hiding but not overriding

```
{
P s v main (String[] args)
{
sopln ("child main");
}
}
```



Java P↵

O/p: Parent main

Java C↵

O/p: child main.

If seems Overriding concept applicable for static methods. But it is not overriding and it is method hiding.

Note: For static methods Overloading and inheritance concepts are applicable but overriding concepts is not applicable. But instead of overriding method hiding concept is applicable.

Inside method implementation if we are using at least one instance variable then that method talks about a particular object, hence we

should declare method as instance method.

Inside method implementation if we are not using any instance variable then this method is not bounded to a particular object, hence we have to declare such type of method as static method irrespective of whether we are using static variable or not.

eg:-
```
class Student
{
    String name;
    int rollno;
    int marks;
    static String cname;

    get_Student_Info()
    {
        return name + "..." + marks;
    }

    getCollegeInfo()
    {
        return cname;
    }

    getAverage(int x, int y)
    {
        return x+y/2;
    }

    getCompleteInfo()
    {
```

Instance Var.

Static Var.

Instance method

Static method.

Static method.

```
    return name+"..."+rollno+"..."+marks+"..."+cname;
  }
}
```

instance method: Because we use atleast one instance variable.

For static methods implementation should be available whereas for abstract method implementation is not available hence abstract & static combination is illegal for methods.

Synchronized modifier:

Synchronized is the modifier applicable for methods and blocks but not for classes & variables.

If multiple threads trying to operate simultaneously on the same java object then there may be a chance of Data inconsistency problem. This is called Race condition. We can overcome this problem by using Synchronized keyword.

If a method or block declared as Synchronized then at a time only one thread is allowed to execute that method or block on the given object so that data inconsistency problem will be resolved.

But the main disadvantage of Synchronized Keyword is it increases waiting time of threads and creates performance problem, hence

if there is no specific requirement it is not recommended to use synchronized keywords.

Synchronized method should compulsory contain implementation whereas abstract method doesn't contain any implementation hence abstract synchronized is illegal combination of modifier for method.

Lecture 39 - Native and transient modifier :-

(1) * **Native modifier** :- Native is a modifier applicable only for methods and we can't apply anywhere else.

The methods which are implemented in non-java (mostly C or C++) are called native methods. or foreign methods.

The main objectives of native keyword are :-

(i) To improve performance of the system.
(ii) To achieve machine level or memory level communication.
(iii) To use already existing legacy non-java code

Pseudo code to use native keyword in java:-

```
class Native
{
    static
    {
        ① Load native library  System.loadLibrary("native library path");
    }
    ② Declare a native method. → public native void m1();
}

class Client
{
    p s v main(String[] args)
    {
        Native n = new Native();
        n.m1(); ③ Invoke a native method.
        :
    }
}
```

For native methods implementation is already available in old languages like C, C++ and we are not responsible to provide implementation. Hence native method declaration should ends with ;.

✓ public native void m1();

✗ public native void m1(){ } CE: native method cannot have a body.

For native methods implementation is already available in old languages. But for abstract methods implementation should not be available, Hence we can't declare native method as abstract, i.e. native abstract combination is illegal combination for methods.

We can't declare native() as strictfp, because there is no gaurantee that old languages follow IEEE 754 standard. Hence, native strictfp combination is illegal combination for methods.

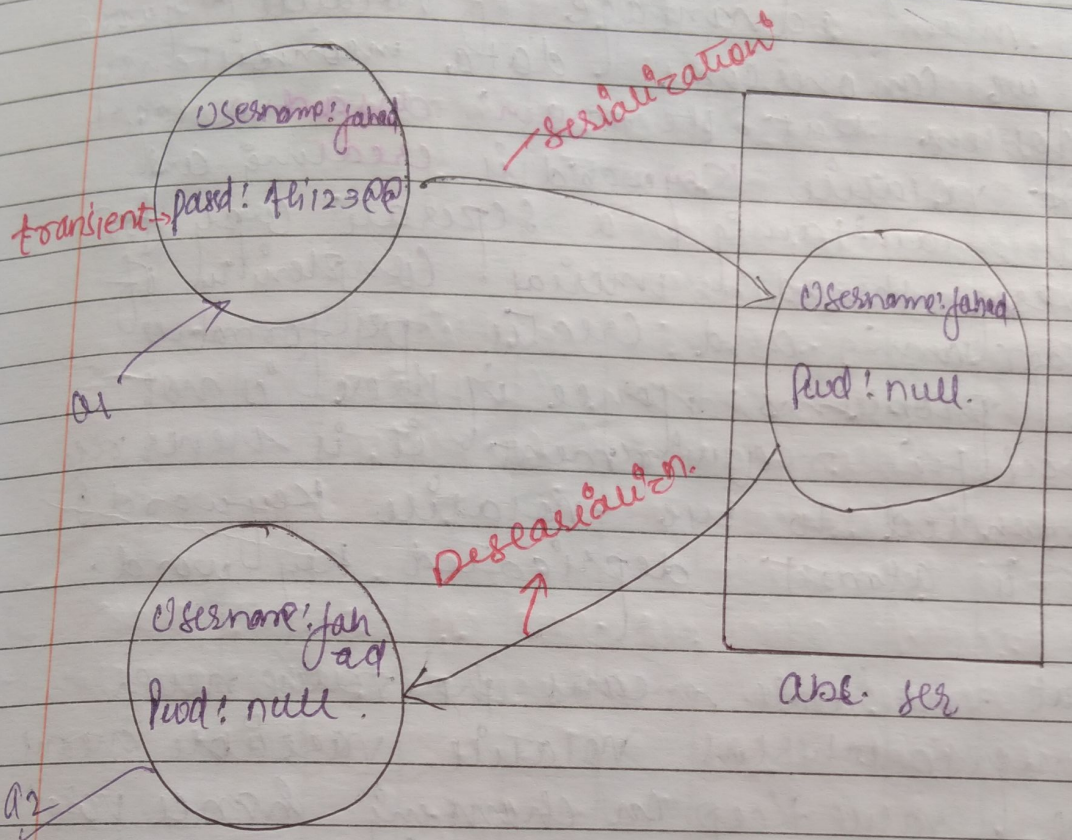
The main advantage of native keyword is performance will be improved. But the main disadvantage of native keyword is it breaks platform independent nature of Java.

(2) Transient Keyword:- Transient is a modifier applicable only for variables.

We can use transient keyword in serialization context.

At the time of serialization if we don't want to save the value of a particular variable to meet security constraints then we should declare the variables as transient.

At the time of serialization JVM ignores original value of transient variables and save default value to the file, hence transient means not to serialize.



(32) Volatile (Modifier) :- Volatile is a modifier applicable only for variables and we can't apply anywhere else.

If the value of a variable keep on changing by multiple threads then there may be a chance of Data inconsistency problem we can solve this problem by using volatile modifier.

If a variable declared as volatile then for every thread JVM will create a seperate local copy

Every modification performed by the thread will take place in local copy so that there is no effect on remaining threads.

The main advantage of volatile keyword is we can overcome data inconsistency problem but the main disadvantage of volatile keyword is creating and maintaining a seperate copy for every thread incrise complexity of programming and creates performance problems, hence if there is not specific requirement it is never recommended to use volatile keyword & it is almost depricated keyword.

final variable means the value never changes, whereas volatile variable means the value keep on changing hence volatile final is illegal combination for variables

* Summary Key Points :

1. The only applicable modifier for local variable is final.
2. The only applicable modifiers for constructors are public, private, protected & default.
3. The modifiers which are applicable only for methods are native.
4. The modifiers which are applicable only for variable are volatile & transient.
5. Two modifiers which are applicable for classes but not for interface final.

volatile | transient

Modifiers	classes				Interface		enum		Constructor
	Outer	Inner	Method	variables	blocks	outer	Inner	outer	Inner	
Public	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
Private	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓
Protected	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓
<default>	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
final	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
abstract	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗
static	✗	✓	✓	✓	✓	✗	✓	✗	✗	✗
synchronized	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗
native	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗
strictfp	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
transient	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗
volatile	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗

Summary Table of all 12 Modifier - According to accessibility

CLASSTIME Pg. No.
Date / /

6. The modifiers which are applicable for classes but not for enum, are, final & abstract.

(5hrs) * Lecture 40 :- Interfaces :- (08/08/22 - Monday)

1) Introduction
2) Interface Declaration & implementation.
3) extends & implements
4) Interface methods
5) Interface variables
6) Interface naming conflicts
   - (i) Method naming conflicts
   (ii) Variable naming conflicts
* 7) Marker Interface
8) Adapter classes
9) interface Vs abstract class Vs Concrete class.
* 10) Difference b/w interfaces and abstract class.
11) Conclusion.

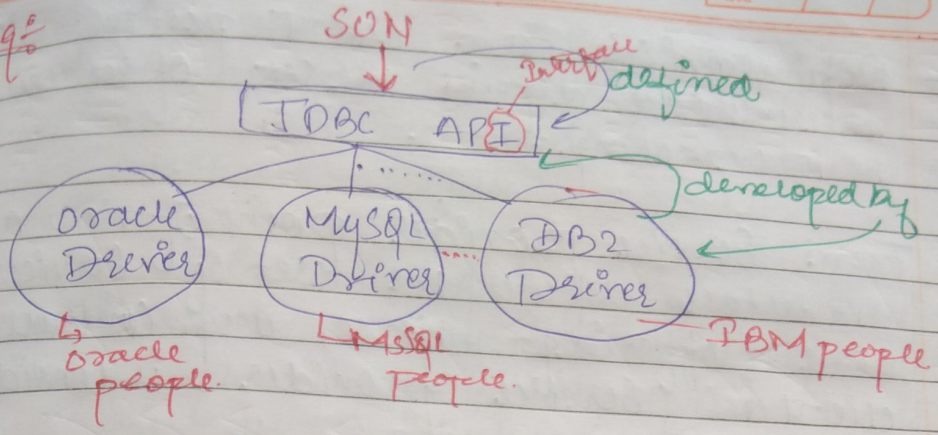
1. Introduction :-

Definition 1 :- Any Service requirement specification. (SRS). is considered as an interface

eg 1 :- JDBC API act as requirement specification to develop database driver. So, database vendors is responsible to implement this JDBC API.

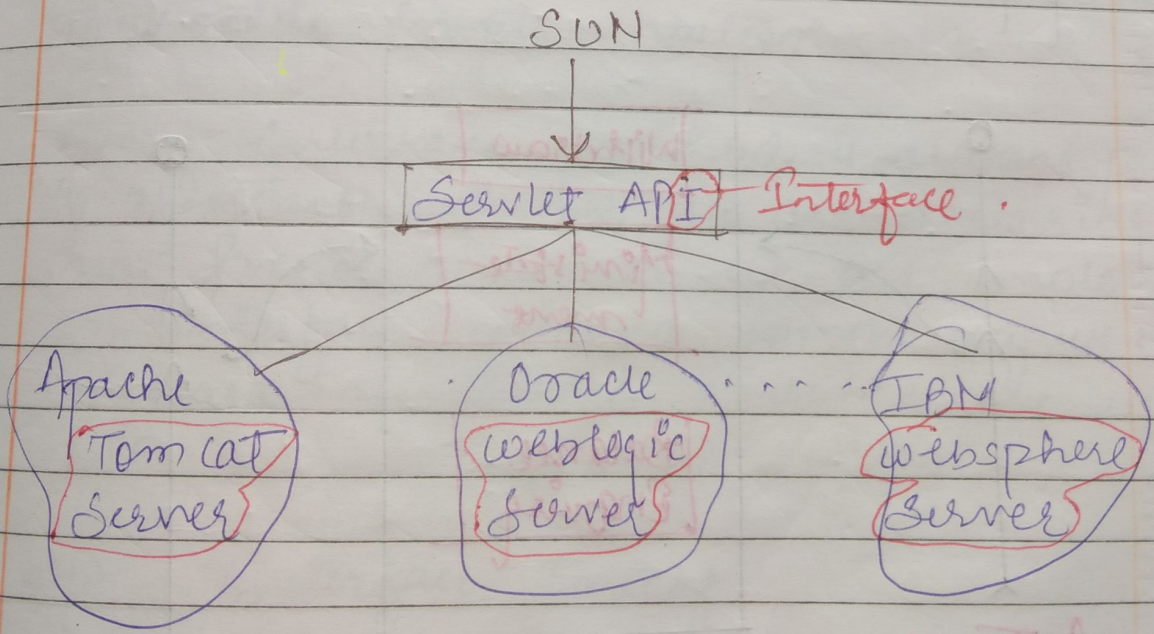
CLASSTIME Pg. No.
Date / /

eg:-



SUN people is responsible to define JDBC API (SRS) and database vendor is responsible to develop JDBC API. (own driver class).

eg2 :- Servlet API act as Requirement specification to develop web server web server vendor is responsible to implement servlet API

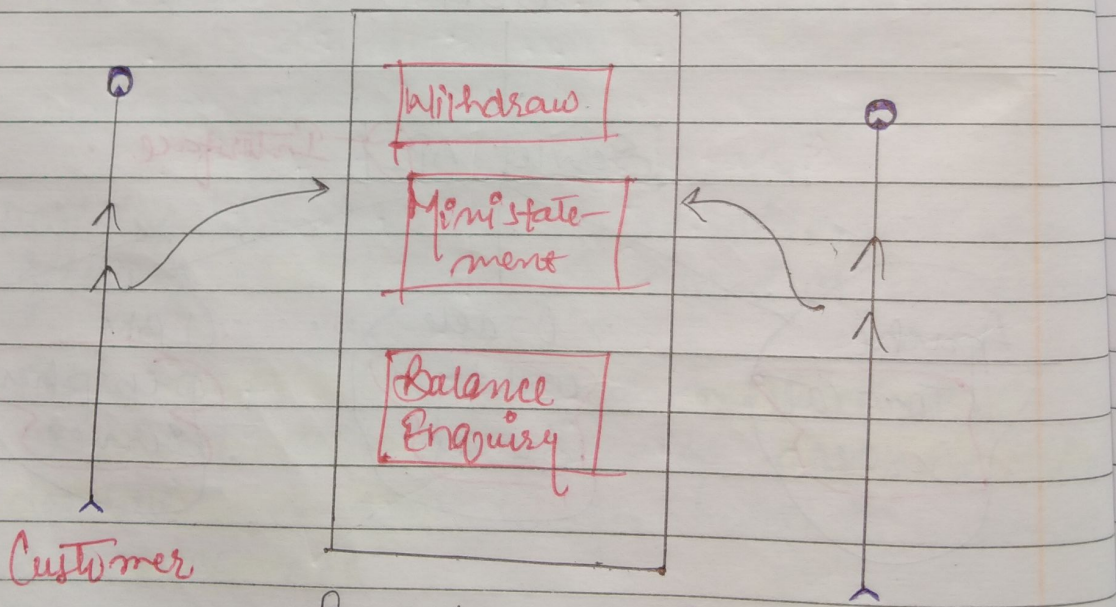


CLASSTIME Pg. No.
Date / /

* Definition 2 :- From client point of view an interface defines the set of services what he is expecting.

From Service provider point of view an interface defines the set of services what he is offering. Hence, any contract b/w client & service provider is considered as an interface.

Through Bank ATM GUI Screen Bank people are highlighting the set of services what they offering at the same time the same GUI screen represents the set of services what customer is expecting; Hence this GUI screen act as contract between Customer & Bank people.



Definition 3:- Inside Interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

Summary definition:- Any Service requirement specification or any Contract b/w client & service provider or 100% Pure abstract class is nothing but interface.

(2) Interface declaration & implementation:-

Whenever we are implementing an interface for each and every method of that interface we have to provide implementation otherwise we have to declare class as abstract then next level child is responsible to provide implementation.

Every interface method is always public & abstract whether we are declaring or not hence whenever we are implementing an interface method compulsory we should declare as public, otherwise we will get CTE.

eg:- Interface Interf
{
    void m1();
    void m2();
}

```
abstract class ServiceProvider implements Inter
{
    public void m1()
    {
    }
}
class SubServiceProvider extends ServiceProvider
{
    public void m2()
    {
    }
}
```

(3) extends Vs implements :-

(i) A class can extend only one class at a time.

(ii) An interface can extend any no. of interfaces simultaneously.

```
interface A
{
}
```

```
interface B
{
}
```

```
interface C extends A, B
{
}
```

✓

So, that's why Java support multiple inheritance.

(iii) A class can implement any no. of interfaces simultaneously.

(iv) A class can extend another class and can implement any no. of interfaces simultaneously.

eg: class A extends B implements C,D,E

{

}

3 ✓

Which of the following is valid?

(i) A class can extend any number of classes at a time. ✗

(ii) A class can implement only one interface at a time. ✗

(iii) An interface can extend only one interface at a time. ✗

(iv) An interface can implement any no of interfaces simultaneously. ✗

(v) A class can extend another class or can implement an interface but not both simultaneously. ✗

(vi) None of the above. True.

Q. Consider the following expression X extends Y.

[X extends Y]

For which of the following possibilities of X&Y the above expression is valid.

(i) Both X and Y should be classes.

(ii) Both X and Y should be interfaces.

(iii) Both X and Y can be either classes (or) interfaces

(c) No restrictions.

(2) [X extends Y, Z]
X, Y, Z should be interface.

(3) [X implements Y, Z]
X → class
Y, Z → interfaces.

(4) [X extend. Y implements Z.
X, Y → class
Z → interfaces.

(5) X implements Y extends Z.
CE: because we have to take extends first followed by interface.

Lecture 41 :-

(4) * Interface methods :- Every method present inside interface is always public & abstract whether we are declaring or not.

Eg:- Interface (Interf)
{
void m1();
}
→ public: To make this method available to every implementation class.

CLASSTIME Pg. No.
Date / /

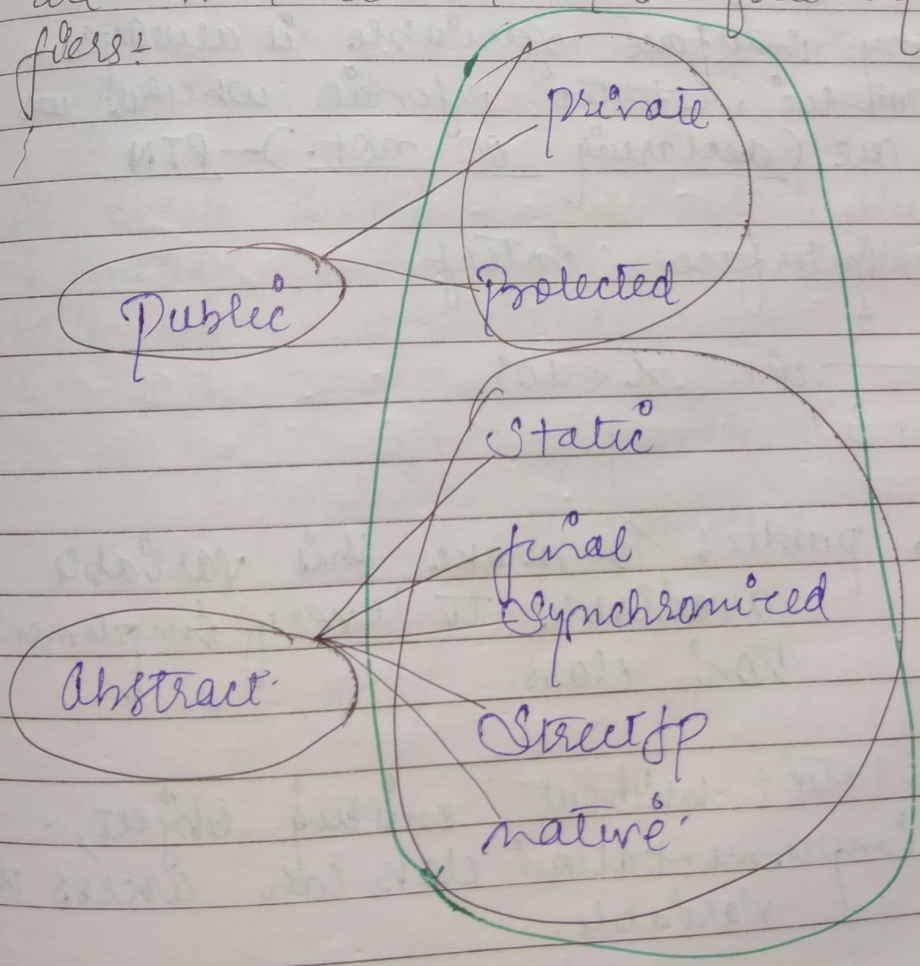
Abstract: Implementation class is responsible to provide implementation.

Q- Since inside interface the following method declaration are equal. ? —

equal
void m1();
public void m1();
abstract void m1();
public abstract void m1();

Ans → Yes equal.

As every interface method is always public and abstract we can't declare interface method with the following modifiers:



Which of the following method declarations are allowed inside interface.

public void m1(){ }; ✗
private void m1(); ✗
protected void m1(); ✗
static void m1(); ✗
public abstract native void m1(); ✗
abstract public void m1(); ✓

(5) Interface variables:- An interface can contain variables, the main purpose of interface variable is - to define requirement level constants.

Every interface variable is always public, static, final whether we are (declaring or not.) - PTN

```
interface interf
{
    int x = 10;
}
```

→ public: To make this variable available to every implementation class.

→ static: Without existing Object, implementation class can access this variable.

CLASSTIME Pg. No.
Date / /

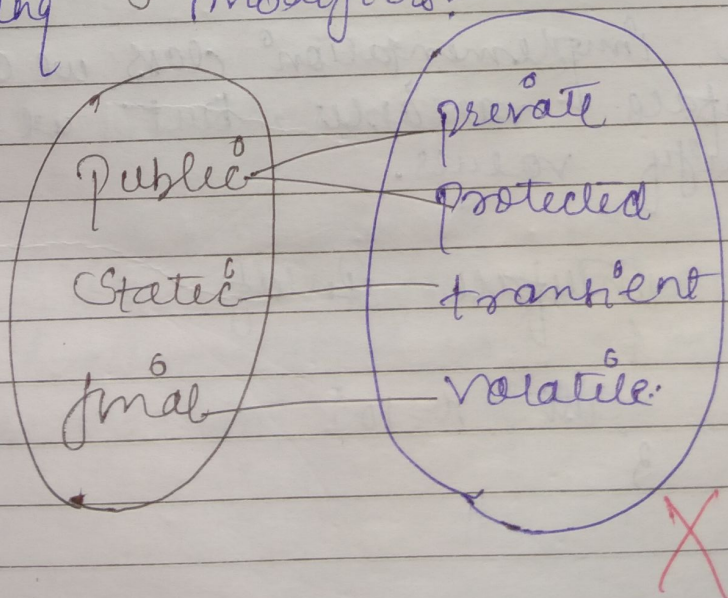
↳ final: If one implementation class changes value then remaining implementation classes will be affected. To restrict this every interface variable is always final.

Hence, within the interface the following variable declarations are equal.

```
int x = 10;
public int x = 10;
static int x = 10;
final int x = 10;
public static int x = 10;
public final int x = 10;
static final int x = 10;
public static final int x = 10;
```

Yes, all are equal.

As every interface variable is always, public, static, final - We can't declare with the following modifiers:



For interface variable compulsory we should provide initialization at the time of declaration otherwise we will get CTE.

```
interface Interf
{
    int x;
}
```

CE:→ = expected.

Inside Interface which of the following variable declarations are allowed.

int x; ✗
private int x=10; ✗
protected int x=10; ✗
volatile int x=10; ✗
transient int x=10; ✗
public static int x=10; ✓

Inside implementation class we can access interface variables, but we can't modify values.

eg:-

```
interface Interf
{
    int x=10;
}
```

```
class Test implements Interf
{
    p s v main(String[] args)
    {
        x = 777;
        sopln(x);
    }
}
```

X

CE: Cannot assign a value to final variable x.

```
class Test implements Interf
{
    p s v main(String[] args)
    {
        int x = 777;
        sopln(x); 777 ← o/p
    }
}
```

(6) Interface naming conflicts :-

(i) Method naming conflicts :-

Case I :- If two interfaces contains method with same signature & same returntype then in the implementation class we have to provide implementation for only one method.

Eg :-

eg:

```
interface Left
{
    public void m1();
}
```

```
interface Right
{
    public void m1();
}
```

```
class Test implements Left, Right
{
    public void m1(){}
}
```

Case II: If two interfaces contains a method with the same name but different argument types then in the implementation class we have to provide implementation for both methods and these methods act as overloaded methods.

eg:

```
Interface Left
{
    public void m1();
}
```

```
Interface Right
{
    public void m1(int i)
}
```

```
Class Test implements Left, Right
{
    public void m1()
    {
    }
    public void m1(int i)
    {
    }
}
```

overloaded method.

except

Case II :- If two interfaces contains a method with the same signature but different return types then, it is impossible to implement both interfaces simultaneously. (if return types are not covariant).

eg :-

```
interface Left
{
    public void m1();
}
```

```
interface Right
{
    public int m1();
}
```

We can't write any Java class which implements both interfaces simultaneously.

If a Java class can implements any no of interfaces simultaneously :-?

Ans :- Yes, It's Except a particular case. :-
If two interfaces contain a method with the same signature but different return types then it is impossible to implement both interfaces simultaneously.

(ii) Interface variable naming conflicts :-

Two interfaces can contain a variable with the same name and there may be a chance of variable naming conflicts. But we can solve this problem by using interface names.

CLASSTIME Pg. No.
Date / /

eg:
```
interface Left                  interface Right
{                               {
  int x=777;                      int x=888;
}                               }

class Test implements Left, Right.
{
  p s v main(String[] args)
  {
    //sopln(x); CE: reference to x is ambiguous.
    sopln(Left.x); 777
    sopln(Right.x); 888
  }
}
```

*Lecture 42 :

(7) **Marker Interface:** If an interface doesn't contain any methods & by implementing that interface if our objects will get some ability such type of interfaces are called marker interface.

eg: Serializable (I)
Cloneable (I)
RandomAccess (I)
SingleThreadModel (I)

} These are marked for some ability

(Or)

Ability Interface.

100)
Tag interface.

eg1:- By implementing serializable interface our objects can be saved to the file and can travel across the network.

eg2:- By implementing cloneable interface our object will provide to produce exactly cloned objects.

Q- Without having any methods how the objects will get some ability in marker interfaces?

Ans- Internally JVM is responsible to provide required ability.

Q- Why JVM is providing required ability in marker interfaces.

Ans- To reduce complexity of programming & to make Java language as simple.

Q- Is it possible to create our own marker interface?

Ans- Yes, but customization of JVM is required.

(8) Adapter classes:- A adapter class is a simple Java class that implements an interface with only empty implementation.

CLASSTIME Pg. No.
Date / /

Eg:

```
interface X
{
    m1();
    m2();
    m3();
    :
    :
    m1000();
}
```

adapter class:

```
abstract class Adapter implements X
{
    m1(){}
    m2(){}
    m3(){}
    m4(){}
    :
    :
    m1000(){}
}
```

(class Adapter — any name)

If we implement an interface for each & every method of that interface compulsory we should provide implementation whether it is required or not required.

```
class Test implements X
{
    m3()
    {
        ≡ 10 lines
    }
    m1(){}
    m2(){}
    m4(){}
    :
    m1000(){}
}
```

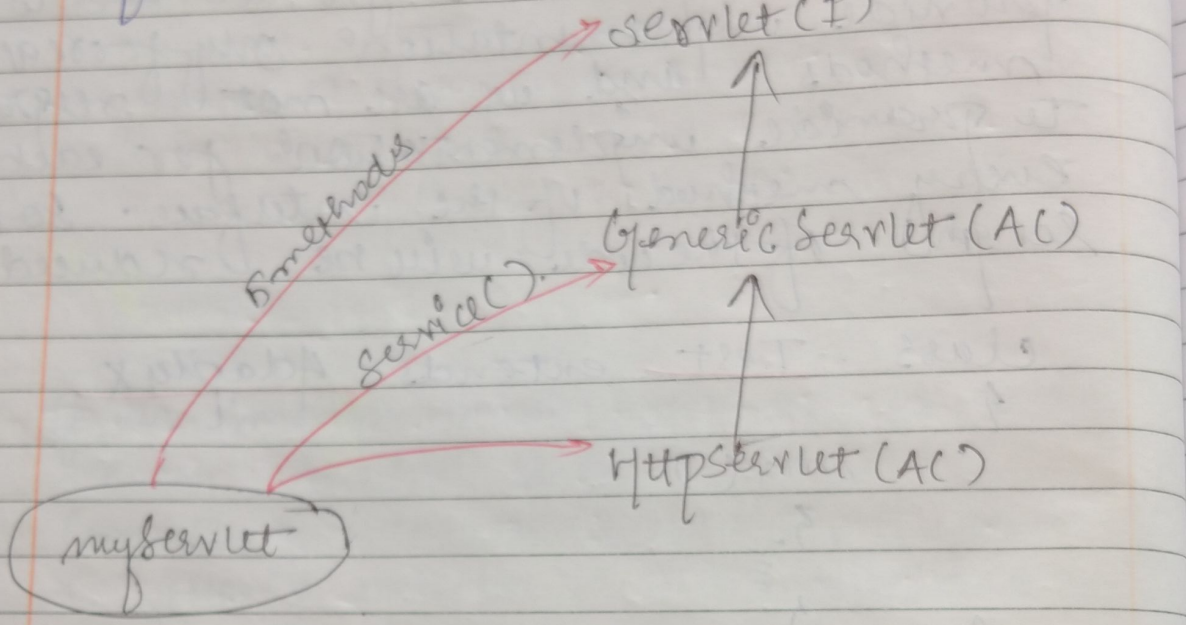
- The problem in this approach is it increases length of the code & reduces readability. We can solve this problem by using adapter classes.
Instead of implementing interface if we extend adapter class we have to provide implementation only for required methods and we are not responsible to provide implementation for each & every method of the interface. So that length of the code will be reduced.

```
class Test extends AdapterX
{
    m3()
    {
    }
}

class Sample extends AdapterX
{
    m4()
    {
    }
}

class Demo extends AdapterX
{
    main()
    {
    }
}
```

CLASSTIME Pg. No.
Date / /

eg:- We can develop a servlet in the following 3 ways:-

(i) By implementing servlet interface (I)
(ii) By extending GenericServlet (AC)
(iii) By extending HttpServlet (AC).



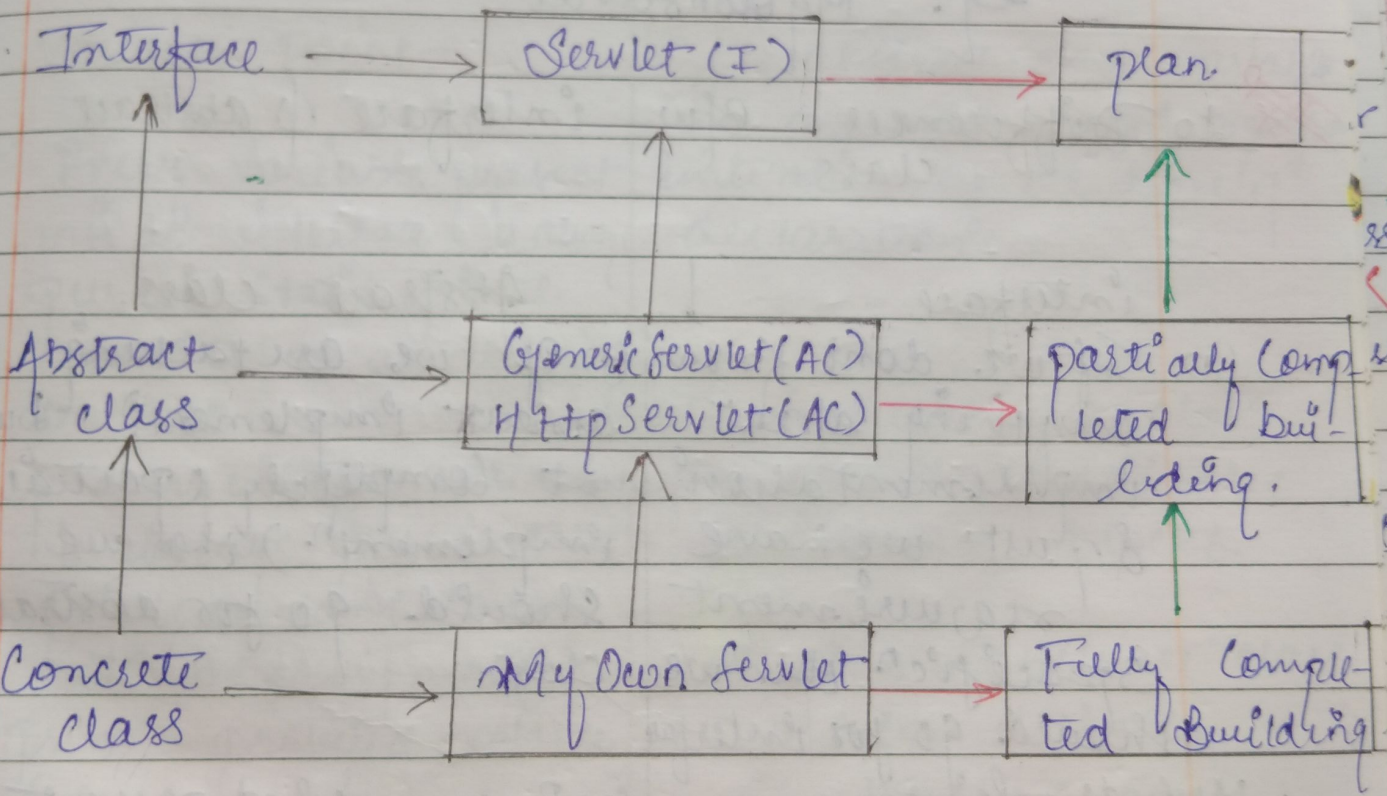
→ If we implement servlet interface for each & every method of that interface we should provide implementation. It increases length of the code and reduces readability.

→ Instead of implementing servlet interface directly if we intend generic servlet we have to provide implementation only for service method(). and for all remaining methods we are not required to provide implementation, hence more or less.

generic servlet act as adapter class for servlet interface.

Note:- Marker interface and adapter classes simplifies complexity of programming. and these are best utilities to the programmer and programmers life will become simple.

Lecture 43 - interface Vs abstract class Vs concrete class.



1. If we dont know anything about implementation just we have requirement specification then we should go for Interface.

eg:- Servlet (I)

If we are talking about implementation but not completely (partial implementation) then we should go for abstract class (AC)

eg: Generic Servlet (AC)
HttpServlet (AC). etc

3. If we are talking about implementation completely & ready to provide service then we should go for concrete class.

eg: MyOwnServlet

10 Differences B/w interface & abstract class

interface	Abstract class.
(1) If we dont know anything about implementation & just we have requirement specification then we should go for interface	(1) If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
(2) Inside interface every method is always public & abstract whether we are declaring or not. Hence interface is considered as 100% pure.	(2) Every method present inside abstract need not be public & abstract & we can take concrete methods also.

- Methods - point.

this above

pure abstract class.

(3) As every interface method is always public & abstract, & hence we can't declare with the following modifiers. private, protected, final, static, synchronized, native, & strictfp.

methods

Variables point-

(4) Every variable present inside interface is always public static final. whether we are declaring or not.

(5) As every interface variable is always public, static final, we can't declare with the following modifiers private, protected, volatile & transient.

(6) for interface variables compulsory we should perform initialization at the time of declaration only. Otherwise we will get CTE.

(3) There are no restrictions on abstract class methods modifiers.

(4) Every variable present inside abstract class need not be public static final.

(5) There are no restrictions on abstract class variable modifiers.

(6) for abstract class variables we are not required to perform initialization at the time of declaration.

→ New

7) Inside abstract class we can declare static & instance blocks.

8) Inside abstract class we can declare constructor.

CLASSTIME Pg. No.
Date / /

(P) Inside interface we can't declare static & instance block.

(P) Inside interface we can't declare constructors.

Loophole:

Q- Anyway we can't create object for abstract class but abstract class can contain constructor. what is the need.

Ans: Abstract class constructor will be executed whenever we are creating child class object to perform initialization of child class object.

approach 1: Without having Constructor in abstract class.

```
abstract class Person
{
    String name;
    int age;
    // 100 properties.
}
```

```
class Student extends Person
{
    int rollno;
    Student (String name, int age ..... 101 properties)
    {
        this.name = name;
        this.age = age;
        :
        :
        this.rollno = rollno;
    }
}
```

100 properties.

Student S1 = new Student (101 properties).

```
class Teacher extends Person
{
    String subject;
    Teacher (String name, int age ... 101 properts)
    {
        this.name = name;
        this.age = age;
        :
        :
        this.subject = subject
    }
}
```

100 properties

100 child classes.

Teacher t = new Teacher (101 properties)

More code, Code Redundancy.

CLASSTIME Pg. No.
Date / /

Approach 2 :- With Constructor inside abstract class.

```
abstract class Person
{
    String name;
    int age;
    :
    100 properties.

    Person (String name, int age ...)
    {                    100 properties.
        this.name = name;
        this.age = age;
        :
        100 lines of code
    }
}
```

This constructor will work for every child object creation.

```
class Student extends Person.
{
    int rollno;
    Student (String name, int age. ...)
                    101 properties
    {
        super (100 properties)
        this.rollno = rollno;
    }
}
Student s1 = new Student (101 properties)
```

CLASSTIME Pg. No.
Date / /

class Teacher extends Person.
{
String subject;
Teacher (String name, int age ....)
{ 101 properties.
super (100 property);
this.subject = subject;
}
}

child class.

Teacher t = new Teacher (101 properties).

less code, code Reusnability.

Note:- Either directly or indirectly we can't create object for abstract class.

(ii) Anyway we can't create object for abstract class and interface but abstract class can contain constructor but interface doesn't contain constructor. What is the reason?

The main purpose of constructor is to perform initialization for instance variables.

Abstract class can contain instance variable which are required for child object. To perform initialization of those instance variables constructor is required for abstract class.

But every variable present inside interface

is always public static finals. whether we are declaring or not. and there is no chance of existing instance variable inside interface. Hence constructor concept is not required for interface.

Whenever we are creating child class object parent object won't be created just parent class constructor will be executed for the child object purpose only.

eg:
```
class P
{
  P()
  {
    sopln(this.hashCode());
  }
}
class C extends P
{
  C()
  {
    sopln(this.hashCode());
  }
}
class Test
{
  P S V main(String[] args)
  {
```

```
C    C=new    C();
Sopln ( C.hashCode() ); 100.
}
}
```

Que:- Inside interface every method is always abstract and we can take only abstract method in abstract class also. Then what is the difference B/w Interface & abstract class. i.e. is it possible to replace interface with abstract class.?

Ans→ We can replace interface with abstract class but it is not a good programming practice this is something like recruiting IAS officer for sweeping activity.

If everything is abstract then it is highly recommended to go for interface. but not for abstract class.

approach 1	approach 2.

abstract class X
{
}
 	
Class Test extends X
{
≡
}
	interface X
	{
	}
	
	class Test implements X
	{
	}

(1) While extending abstract class it is not possible	While implementing interface we can extend some

to extend any other class, hence we are missing inheritance Benifit.	other class, Hence we won't miss any inheritance benifit.
2) In this case object creation is costly. eg:- Test t = new Test(); (2 mins)	2) In this case object creation is not costly. eg: Test t = new Test(); (2 sec)

Doubt lecture Q1:-

Lecture 44 - (new VS ~~new~~ Constructor) :-

1. The main objective of new operator is to creat an object.

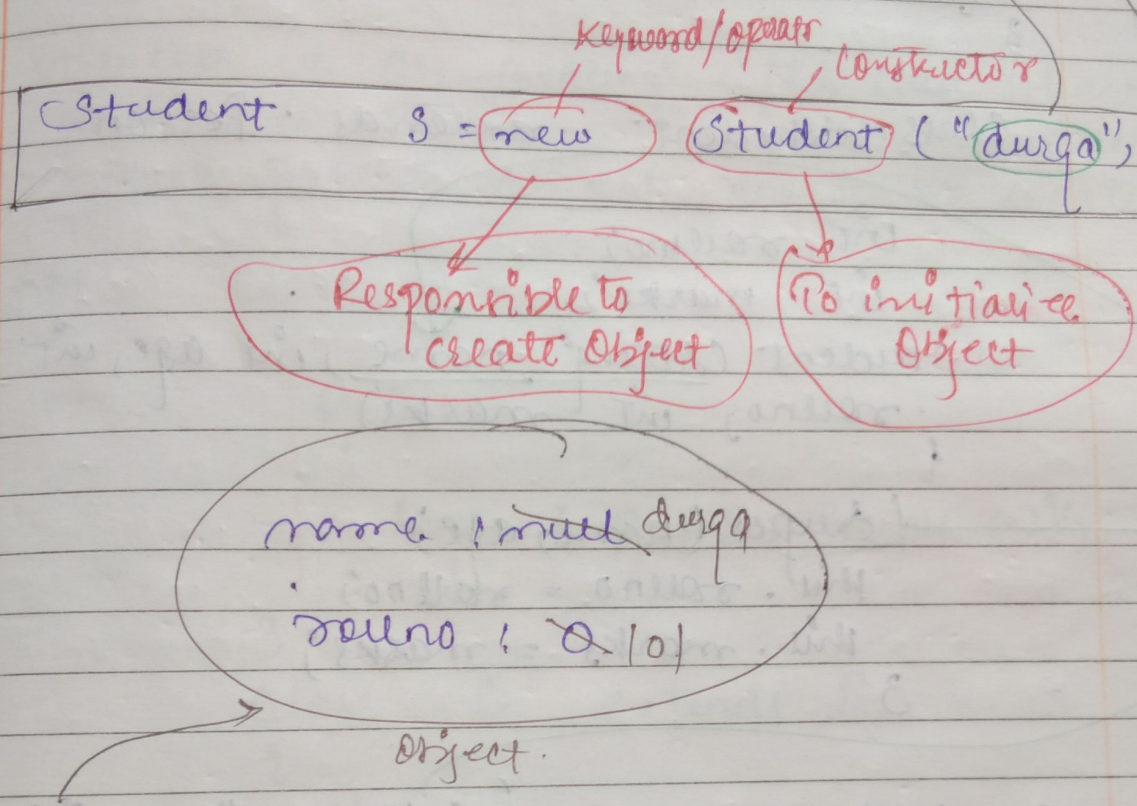
2. The main purpose of constructor is to initialized that object.

3. first object will be created by using new operator and then initialization will be performed by constructor.

eg: class Student
{
String name;
int rollno;
instance variable
Constructor Student(String name, int rollno)
{
this.name = name;

CLASSTIME Pg. No.
Date / /

```
    this.rollno = rollno;
  }
}
```



(3)* Lecture 45 :- Whenever we are creating child class Object automatically parent constructor will be executed to perform initialization for the instance variable which are inheriting from parent.

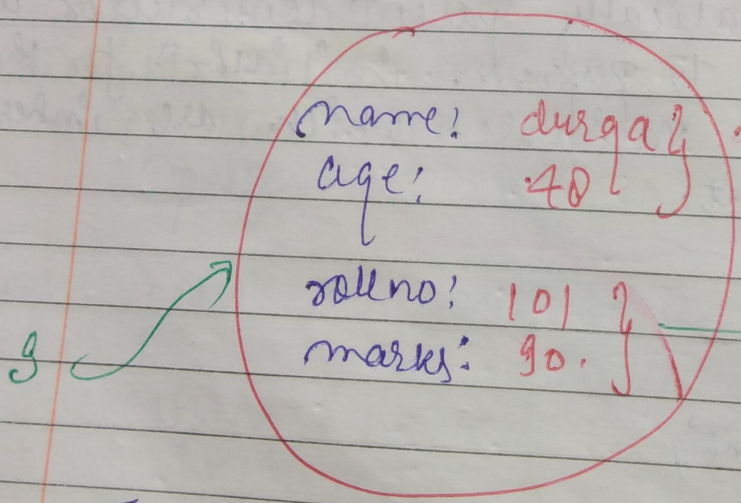
```
Class Person
{
  String name;
  int age;

  Person(String name, int age)
  {
  }
```

```
    this.name = name;
    this.age = age;
}
}
```

Class Student extends Person

```
int rollno;
int marks;
Student(String name, int age, int rollno, int marks)
{
    super(name, age)
    this.rollno = rollno;
    this.marks = marks;
}
```

```
Student s = new Student("durga", 40, 101, 90);
```



In the above program both parent & child constructors executed for child object initialization only.

CLASSTIME Pg. No.
Date / /

Lecture 46 :- Q-2 Whenever we are creating child class object whether Parent object will be created or not ???

Whenever we are creating child class object parent constructor will be executed, but parent object wont be created.

eg:-

```
Class P
{
    P()
    {
        System.out.println(this.hashCode());
    }
}
Class C extends P
{
    C()
    {
        System.out.println(this.hashCode());
    }
}

Class Test
{
    public static void main(String[] args)
    {
        C c = new C();
        System.out.println(c.hashCode());
    }
}
```

In the above example we just created only child class object, but both parent & child constructor executed for that child class object.

Lecture 47:

Q-4 By way we cannot create object for abstract class either directly or indirectly, But abstract class can contain constructor. what is the need?

The main objective for abstract class constructor is to perform initialization for the instance variables which are inheriting from abstract class to the child class.

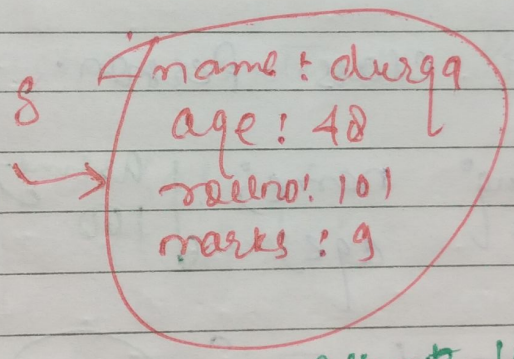
Whenever we are creating child class object, automatically abstract class constructor will be executed to perform initialization for the instance variable which are inheriting from abstract class (code reusability).

Without constructor in abstract class

```
abstract class Person
{
    String name;
    int age;
}
```

} — Assume 100 properties.

```
class Student extends Person
{
    int rollno;
    int marks;

    Student(String name, int age, int rollno,
                                  int marks)
    {
access/act  parent property { this.name = name;
                              this.age = age;
                              this.rollno = rollno;
                              this.marks = marks;
    }

Student s1 = new Student("durga", 40, 101, 9);
```

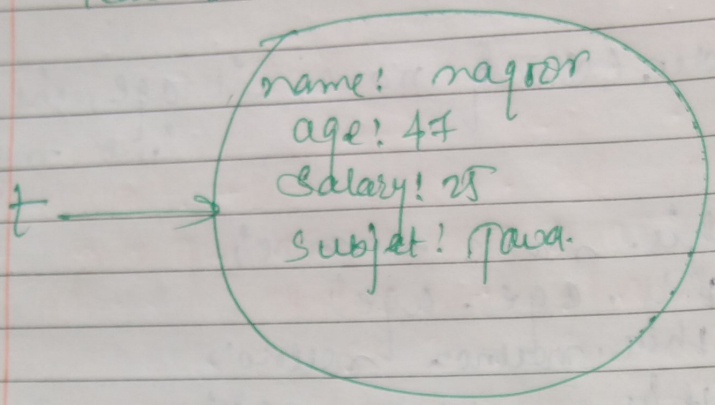


assume 1000 child classes

```
class Teacher extends Person..
{
    double salary;
    String subject;

    Teacher(String name, int age, double salary, String sub)
    {
        this.name = name;
        this.age = age;
        this.salary = salary;
```

this.subject = subject;
}
}

Teacher t = new Teacher("naqoor", 47, 25, "java");



With abstract class constructor

```
abstract class Person.
{
    String name;
    int age;
```

Assume 100

```
Person(String name, int age)
{
    this.name = name;
    this.age = age;
}
}
```

```
Class Student extends Person.
{
    int rollno;
    int marks;
```

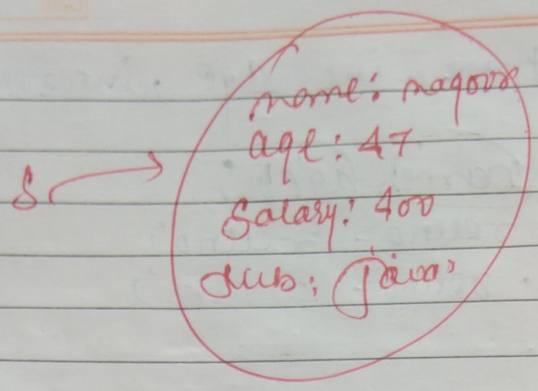
```
Student (String name, int age, int rollno, int marks) {
    super (name, age);
    this.rollno = rollno;
    this.rollno = marks;
}
}
```

Student s = new Student ("durga", 40, 101, 90);

s → name: durga
age: 40
rollno: 101
marks: 90

```
class Teacher extends Person.
{
    double salary;
    String subject;

    Teacher (String name, int age, double salary,
             String subject)
    {
        super (name, age);
        this.salary = salary;
        this.subject = subject;
    }
}
```

Teacher t = new Teacher ("nagoor", 47, 20, "java");



✱ Lecture 48 :-

Que:- Anyway we cannot create object for abstract class and interface. Abstract class can contain constructor but interface does not. why ???

Ans:- The main purpose of constructor to perform initializn of an object. i.e. to perform initializn. for instance variables.

Abstract class can contain instance variables which are required for child class object to perform initializn for these instance variables constructor concept is required for abstract classes.

Every variable present inside interface is always public, static, final whether we are declaring or not. hence, there is no chance of existing instance variables inside interface.

because of this constructor concept not required for interface.

Eg:
```
abstract class Person {
    String name;
    int age;

    Person (String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```
→ Current child class object.

```
interface Interf
{
    int x = 10;
    public
    static
    final
}
```

Lecture 49 :

Qu: If everything is abstract it is highly recomended to go for interface but not abstract class. we can replace interface with abstract class, but it is not a good programming practice (this is something like securities officer for sweeping purpose).

CLASSTIME Pg. No.
Date / /

interface X
{
✓
}

(i) While implementing interface we can extend any other class. Hence we won't miss inheritance benifit.

eg Class Test extends A implements X
{
✓
}

(ii) In this case Object creation is not costly.

eg- Class Test implements X
{
}

Test t = new Test();

2min

abstract class X
{
}

(i) While extending abstract class we can't extend any other class. Hence we are missing inheritance benifit.

eg Class Test extends X, A
{
✗
}

(ii) In this case object creation is costly.

Class Test extends X
{
}

Test t = new Test();

20min

Inside we can take only Abstract methods. But in abstract class, also we can't only Abstract methods. based on our requirnt. Then what is the need of interface? i.e. Is it

possible to replace interface concept with abstract class?

Lecture 50

(1) The purpose of constructor is to create an object. ✗ false
(2) The purpose of constructor is to initialize an object Not to create object. ✓ True
(3) Once constructor completes then only object creation completes. ✗ false
(4) First object will be created and then constructor will be executed ✓ True
(5) The purpose of new keyword is to create object and the purpose of constructor is to initialize that object. ✓ True
(6) We can't create object for Abstract class Directly but indirectly we can create ✗
(7) We can whenever we are creating child class object Automatically Parent class object will be created Internally. ✗ false
(8) Whenever we are creating child class object Automatically Abstract class constructor will be executed. ✓ True
(9) Whenever we are creating child class object: Automatically parent object will be created. ✗ false
(10) Whenever we are creating child class object Automatically Parent Constructor will be executed but parent object won't be created. ✓
(11) Either directly OR indirectly we can't create object for abstract class and hence Constructor concept is not applicable for abstract class. ✗ false
(12) Interface can contain Constructor. ✗ false.